Time Budgeting in a Wireplanning Context

Jurjen Westra Dirk-Jan Jongeneel[†] Ralph H.J.M. Otten Chandu Visweswariah[‡]
Fac. of EE, Eindhoven University of Technology, The Netherlands
[†]Fac. ITS, Delft University of Technology, The Netherlands
[‡]IBM Thomas J. Watson Research Center; this work was performed while at Eindhoven University
jwestra@ics.ele.tue.nl

ABSTRACT

Wireplanning is an approach in which the timing of input-output paths is planned before modules are specified, synthesized or sized. If these global wires are optimally segmented and buffered, their delay is linear in the path length and independent of the position of the modules along these paths. From timing requirements, the total budget left to modules after allocating the appropriate delay to the wires can be determined. This paper describes how this budget can be optimally divided amongst the modules. A novel, static timing-like, mathematical programming formulation is introduced such that the total module area is minimized. Instead of only the worst delay, all pin-to-pin delays are implicitly taken into account. If area-delay tradeoffs are convex, a reasonable approximation in practice, the program can be solved efficiently. Further, efficiency of different formulations is discussed, and a low-cost method of making the budget relatively immune to downstream uncertainties and surprises is presented. The efficiency of the formulation is clear from benchmarks with over 2000 nodes and 5e19 paths.

1. Introduction

In the System-on-a-Chip (SoC) era, big chips consist of many large modules. This partitioning is due to the use of IP, and/or the part of a hierarchical design methodology. Independent concurrent design of modules is a good way to boost designer productivity, or to avoid tool capacity limitations. The problem, however, is that top-level timing constraints have to be translated to time budgets for modules.

Currently, time budgets are only assigned after (partial) physical design of modules. Problems are resolved by iterating between time budgeting and performance tweaking. This can be a time-consuming and cumbersome process without any guarantee on convergence, or even feasibility.

In DSM technology, focus has shifted from gate delay to interconnect delay. Especially delay of long interconnect may become increasingly dominant, and (arguably) worse, unpredictable. This trend also poses problems for timing convergence.

One of the proposed methodologies to tackle the described problems is known as *wireplanning* (see [1, 2] and below). The essence of this methodology is that the uncertainty regarding wire delay, the very cause of the iterations, is removed. In a wireplanning context, these delays, caused by so called global wires, are *planned*. The question of how to distribute a remaining time budget over the modules, however, has not been solved yet, and is the subject of this paper.

First, we will introduce the concepts of wireplanning. Next, we present a new mathematical programming formulation that enables us to perform optimal time budgeting, taking wire delay into account *before* positional information regarding the modules is known. In section 4 we will show how the formulation can be made even more efficient. Section 5 shows how to make the time budgets "robust" with respect to downstream uncertainties. We will present some numerical results, and will conclude with a discussion and conclusions.

2. Wireplanning

The problem of how to organize placement and routing in chip design is a classical one. Obviously the two have significant impact on each other. Traditionally placement precedes routing, though some interleaving has been tried in the past. With the dominating role of wiring in the performance of integrated systems, more and more tool developers started to think of "wiring-first" approaches. The term *wireplanning* was already coined in the eighties to reflect that idea in the context of a data-path synthesizer. Nowadays *wireplanning* stands for an approach introduced in [1] that first allocates parts of the time budget to the wiring, before fixing, synthesizing, sizing and placing the modules.

In this paper we consider the task of *planning* a *functional network*[1] inside a given footprint while constraining the delays between inputs and outputs of the network. The footprint prescribes both the geometrical shape as well as the positions of the terminals along the boundary. Although manipulating the network[2] to ensure certain properties is arguably

[1] *Functional networks* are defined in [1] as acyclic directed graphs whose nodes represent functions and whose arcs indicate communications. This means that cycles are either local to the function, closed outside the network or previously broken in a sensible way, such as at latch boundaries.
[2] These manipulations form a framework for decomposing, absorbing and duplicating functions to obtain equivalent networks. A complete algebra for performing this at the logic synthesis level by monotonicity-preserving transformations has been worked out in [12].

part of the wireplanning task, we assume the functional network here to be fixed. Further, this network is *monotonic*, i.e. there exists a point placement that causes no detours in input-output paths.

Wireplanning as in [1] rests on two observations:

1. If the size of the individual modules is small compared to the total area, then every functional network has an equivalent network with a (near-)monotonic placement.
2. If the wires between the modules are optimally segmented and buffered, then they have a delay linear in their length.

The former property is the consequence of the fact that any network with a single node containing all functionality at the site of a primary output has a monotonic placement. Of course this is seldom a desirable realization, but the latter property allows for extracting functionality and moving it over wire segments without impacting delay.

2.1 Methodology

One of the ideas behind wireplanning is the concept of *stepwise refinement*. Since it is not possible to formulate and solve an optimization for the whole design cycle, it is split up in separate steps with different optimizations. More and more information about the (final) design is frozen, and since wireplanning is non-iterative, more and more parts of the design space are decisively pruned off very early in the flow, effectively limiting design space and design iterations. Once a decision is taken, based on the available information *at the time*, this decision is final. In our context, this means that the time budgets we calculate are "golden." Calculated areas have the status of *estimates*.

For the implementation of (sub)modules, *fixed delay synthesis* [9, 8] is employed. Gate sizes are assigned in a systematic way, based on the observation that if gates are scaled with their output capacitance, the delay stays the same. Loosely formulated, the result is that *any delay can be met* at the price of area, respecting some parasitic delay. Obviously, sometimes different architectures are more efficient for different constraints, but the statement remains essentially the same.

To see how our time-budgeting technique may be used, we sketch a wireplanning flow conformant to [1]. At the functional level, we have no more than a network in terms of functional nodes and interconnect. Wireplanning tools should aid in analyses and proposals for duplication, absorption and decomposition. Key is that monotonicity is maintained. The choice of the footprint (area and pad positions) is of importance here, and should be incorporated. In this abstraction, there is little awareness of area of nodes. This reflects the idea of stepwise refinement, and is the way complexity is handled.

After this, our time budgeting approach may be employed. As a side effect, we get area estimates, which may result in a "no-go" decision if the footprint does not provide enough area.

Now, floorplanning should result in relative positions of modules. After this stage, more or less classical logic/physical synthesis and physical design can be employed. With the now known areas, placement can be performed, based on the floorplanning, followed by toplevel synthesis.

2.2 Assumptions

The time budgeting we perform in this paper relies on the availability of speed-area trade-offs for each of the individual modules. In section 3.2 we discuss how they may be obtained. For now we will just assume they are available.

Our optimization is based on pin-to-pin timing constraints. In the simplest case, they can be obtained by subtracting timing assertions for output and input pins, but more general, they can be specified as arbitrary pin-to-pin required delays.

3. Problem formulation

First, we will more formally introduce the problem we solve, and formulate it in a way fit for mathematical program solvers.

Problem: Given tradeoffs between area and speed for each module, a footprint for the chip as a whole, a functional network, certain technology dependent parameters, and pin-to-pin timing requirements, find the time budgets for each macro such that the total macro area is minimized, and timing requirements are met.

In a wireplanning flow, time budgeting is performed very early. Timing should be closed, rather than optimized, making area minimization a natural choice. This has the advantage that footprint evaluation can be done very early in the flow. Also, area usually correlates well with power. Further, physical design is easier if more area is available for wiring. Note that especially for ASICs, our formulation with timing constraints rather than area constraints is natural.

3.1 Timing constraints

The delay of a pin-to-pin path is the sum of alternating wire and module delays. In a wireplanning flow we can estimate the total wire delay of the path *before floorplanning* (see Fig. 1).

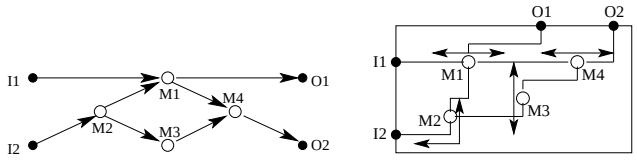

Figure 1: A functional network, and a monotonic wireplan for it. Path wiredelay is fixed regardless of the chosen monotonic wire topology and positions of modules along the wires.

We combine linearity and monotonicity: the wire delay W_j of segment j equals its length L_j times some technology/layer dependent constant c. If a layer assignment is not yet available, an average for the layers available for global wiring is a reasonable approximation. Monotonicity assures that the

length of a path p, the sum of segment lengths, equals the manhattan distance L_{io} between the primary pins i and o. With this, the path wire delay becomes: $W_p = \sum_{j \in p} W_j = \sum_{j \in p} c \cdot L_j = c \sum_{j \in p} L_j = c \cdot L_{io}$, and can be calculated from the footprint; it becomes essentially a constant. Note that the derivation does not hold for unbuffered wires with their quadratic delays.

Now we can formulate the path delay for any path p from I to O as the sum of module delays $d_M^{mn}(A_M)$, functions of the area A_M of the module M, and some constant wire delay W^{IO}, and the timing requirement reads as:

$$D_p = \sum_{M \in p} d_M^{mn}(A_M) + W^{IO} \leq T_{req}^{IO}, \qquad (1)$$

where m and n are the pins of M the path goes through.

In practice, the derivation above means the position of modules along a path needs not to be fixed for our approach. In fact, we do not even need a wire topology, as long as the (subsequently) chosen topology respects monotonicity. This is what makes it possible to conduct time budgeting this early in the flow.

3.2 Tradeoff modeling

Mathematical optimizers need us to specify values and gradients (approximations) for the relation between area and pin-to-pin delays of a module. Usually, with experienced designers, there is an awareness of this tradeoff, resulting in points in the design space. Another source is *Design Space Exploration*[11]. Further, quick runs of logic synthesis will give an idea of the tradeoff. Finally, the relation between area and delay with respect to gate/transistor sizing is well understood.

If we now prune the design space for Pareto points, we can fit a mathematical function to these points. We have chosen a subclass of so-called *posynomial functions*, shown in (2). The fitting results are good, and with a simple transformation, these functions become convex, and convex programming techniques with their guarantee of global optimality can be used.

$$f(t) = \sum c_j \prod {t_i}^{a_{ij}} \quad \text{with} \quad c_j \geq 0, a_{ij} \in \mathbb{R} \qquad (2)$$

In an example below, it is shown that only low-order functions and only a few data points are needed for accuarate modeling. Simple bounds should be used to ensure the valid part of the curve is used. Note that our formulation does not restrict us to use these functions. In the general case, however, we have to resort to nonlinear programming techniques with only a guarantee of local optimality.

3.2.1 Example. We took an industrial example from [11] of which the design space was extensively explored. The design is an embedded system with a VLIW core, cache and a non-programmable systolic array. Parameters such as the number of integer and floating point units and all kinds of registers are varied. We took the results from Fig. 13 of this report: Overall System Pareto (VLIW + memory). This curve depicts the trade-off between area and performance.

We took a small number of points from this graph, and fitted it to the formula $y = a/(x + b) + c$, which can be made posynomial by the simple transformation $x' = x + b$ (note that this only adds a constant to the object function, and does not influence the optimization). The results are shown in Fig. 2, and as can be seen, the fitting is very good.

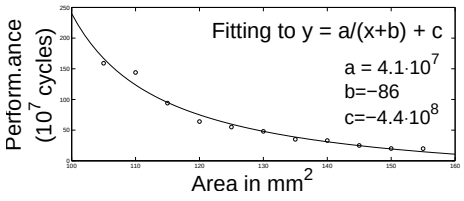

Figure 2: A fitting on an area-performance trade-off. A simple transformation makes the formula posynomial.

3.3 Mathematical programming formulation

The simplest formulation would be to enumerate all paths in the circuit, constrain each to be smaller than its respective timing constraint, and minimize for area. For circuits of practical size however, the number of paths explodes, and the problem becomes infeasible; therefore we introduce a new, more efficient formulation.

As in [3], we construct a *timing graph*. Nets of the functional network become nodes, and directed arcs from nodes m to n depict delay variables $d_M^{mn}(A_M)$; generally non-linear functions of the area A_M of the associated module M (our tradeoff). As illustrated by Figs. 3 and 4, at each node n in the timing graph, we introduce variables AT_n^I representing the late-mode arrival time at n considering only paths which originate at primary input I.

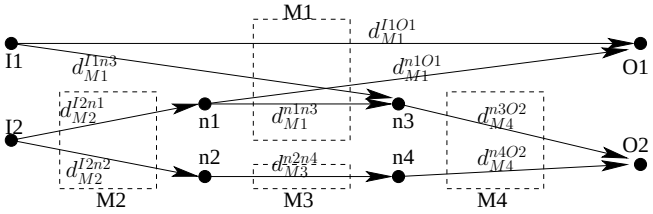

Figure 3: The timing graph for Fig. 1 with the delay variables on the arcs. Modules are drawn with dashed line.

Our approach resembles the well-known *static timing* approach of for instance [10, 3]. The difference is that in these works only *one* arrival time variable is maintained at each node[3], while we maintain such a variable for *each primary input in the nodes input cone* (see Fig. 4). The reason we do this is that this way, we can address different wire delays for different input-output paths. We can take *all* input-output

[3]Technically, there is one each for rising and falling signals, which we can also accomodate depending on the sophistication of the timing models.

delays into account, instead of the worst only. Note that our approach does not increase the complexity of the problem as compared to static timing: at each node at most $|PI|$ (the number of primary inputs) variables are introduced. In practice this number is far lower.

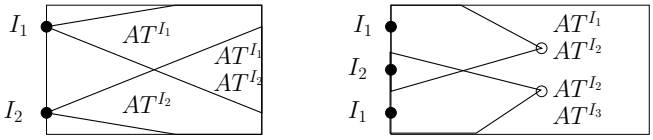

Figure 4: AT variables for a primary input exist at a node if the node is in the fanout cone of the input (left), or equivalently, if the input is in the fanin cone of the node (right).

Now we can formulate a mathematical program:

$$\text{minimize} \quad \sum_{M \in Modules} A_M$$

$$\text{subject to}$$

$$\begin{array}{ll} AT^I_n \geq AT^I_m + d^{mn}_M(A_M) & \forall I \in PIC(m), e(m,n) \in E \\ AT^I_O \leq T^{IO}_{req} - W^{IO} & \forall I \in PI, O \in PO, \end{array} \tag{3}$$

where E, PI, PO and $PIC(m)$ are the sets containing all edges, primary inputs, primary outputs and the intersection of PI and the input cone of m, respectively, and $e(m,n)$ is an arc from m to n. We can add simple bounds on the area of individual modules:

$$A_{min,M} \leq A_M \leq A_{max,M}. \tag{4}$$

These kind of bounds can result in *slack separation* that may be used later on (see section 5). The formulation above can be fed to a mathematical program solver. The shape of the area-delay tradeoff d^{mn}_M decides what kind of solver can be used. If the trade-offs have convex shapes, or can be made convex through a transformation, convex program solvers with their guarantee of global optimality and efficient implementation may be used.

4. Enhancing efficiency

The formulation of the previous section is already orders of magnitude better than a straightforward formulation that enumerates all paths. In this section, we will introduce techniques to reduce runtime and increase feasible problem size further.

4.1 Forward vs backward formulation

Untill now, we have have traversed the timing graphs from input to output, stating that the AT at a node should exceed the AT at its predecessor by the module delay. We can however also work with *Required Arrival Times* (RATs). The constraints will look similar, except that ATs refer back; they have a superscript denoting a primary input, while RATs are with respect to *primary outputs*: they have a primary output as superscript. ATs ripple forward through the network from the inputs; RATs ripple backward from the outputs, hence the names *forward*, and *backward formulation*. If one reverses all directed edges in the network, its forward formulation equals the backward formulation of the original network. Both formulations have the exact same solution, but their efficiencies may differ.

Consider a node n. If it is connected to far more primary inputs than to primary outputs, it is (locally) advantageous to use the RAT formulation since there will be far less RAT than AT variables.

It is easy to find the total number of constraints with $d^{ab}_M(A_M)$ variables: it equals the total number of arcs. The number of variables at a node n is the number of primary inputs in its fanin cone ($|PIC(n)|$). Now we find the number of constraints with n on the lefthand side $N(n)$:

$$N(n) = \sum_{m \in pred(n)} \#edges(m,n) \cdot |PIC(m)| \tag{5}$$

where $pred(n)$ denotes the set of direct predecessors of n. We can obtain the total number of constraints by simply summing over all nodes, and adding the number of timing constraints ($\sum_{o \in PO} |PIC(o)|$).

If we do the same for the backward formulation, we can compare the numbers of both formulations, and pick the best one. As a rule of thumb, it is best to pick the backward formulation if there are fewer primary outputs than inputs. Note that in the static timing formulations of [10, 3] both formulations are the exact same, and there is nothing to be gained.

4.2 Pruning

In [4] a technique called *pruning* to reduce problem size, degeneracy and redundancy without sacrificing accuracy was introduced. As illustrated by Fig. 5, the basic pruning operation is a graph transformation that replaces a node with touching arcs, and replaces it with other arcs. The variables on the arcs are such that the associated optimization problem is equivalent to the original one.

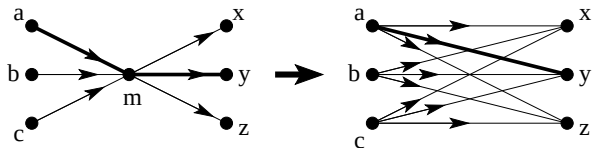

Figure 5: The basic pruning operation with constraints for the bold path.

Generally, solver performance depends on the number of constraints, the number of variables, and the total number of terms in the constraints. Pruning impacts these numbers. It is possible to assign to each node a *gain*, a measure of the benefit in case this node would be pruned, taking these effects into account. If for example two constraints with four variables may be replaced by one constraint with six variables, this may or may not be beneficial, depending on the used solver. This is reflected by a positive or negative gain.

Nodes are greedily pruned until no nodes with positive gains exist anymore.

In [4], only the numbers of variables and constraints are taken into account, and given equal weights. We also considered the number of terms, and tailored the associated weights to the solver we used.

In the original formulation in [4] only one AT variable resides at each node, while in our approach one for each primary input may exist. Therefore, we have adjusted the pruning procedure to calculate gains for and prune AT *variables* rather than nodes. This way, an AT at node n may be pruned for primary input i_1, but not for primary input i_2. Another way of looking at this is that *for each primary input*, we construct a timing graph consisting of the input and its fanout cone, and apply the original pruning procedure. Then, the resulting optimization program is simply the sum of object functions and the concatenation of the associated constraints.

In our case, the pruning procedure requires only one graph traversal, and results not only in a dramatically more compact formulation, but also one that is numerically better-conditioned.

5. Introducing robustness

The time budgets we assign are calculated based on assumptions on path lengths, monotonicity and tradeoffs. In real life, however, some of these assumptions may be hard to meet later on: one may for instance sometimes need to deviate from monotonicity. Another uncertainty is the accuracy of the tradeoff models. More generally, one can say that in a non-iterative designflow, one needs a certain amount of "slack separation"[4] in order to obtain "robustness" with respect to uncertainties later on. Here we will show how slack on the majority of paths is introduced at very low cost.

The formulation itself ensures a certain amount of slack: wire delay is calculated as the path length times some constant. When the modules are realized, however, they will occupy space, effectively reducing wire length, and thus delay. Secondly, our formulation ensures that all input-output pairs will have a critical path: a path whose delay equals the constraint. This implies that every module is on a critical path. It does however, not imply that every wire segment is. This offers the possiblity of detouring these wires. Thirdly, if simple bounds on area are used, this may also result in slack. Finally, wire delay is calculated for ideally buffered wires. If logic is pulled out of the module, and spread over the wires, buffers may be replaced by "useful" gates. Therefore, our wire delay estimation is conservative.

5.1 Formulation with penalty function

The slack that is inherent in our formulation may not be enough, therefore we introduce a way to provide slack at little area cost. This also has the advantage that *truly critical paths*, paths that limit circuit performance most, are revealed. It becomes clear where the main thrust of the design effort should focus.

The numerical optimizer will slow paths down to gain very little area. In our context, this is undesirable. In [5] observations similar to ours were made in the context of circuit optimization. Although [5] aims at speed optimization, we can follow its line of thought. The idea is that for each input-output pair, we add a penalty function to the object function, putting *downward pressure* on its ATs:

$$\min \sum_{m \in modules} A_m + k \sum_{O \in PO} \sum_{I \in PIC(O)} P(T^{IO}_{req}, AT^I_O). \quad (6)$$

The penalty function $P(T_{req}, AT)$ should decay to zero quickly when there is sufficient separation. For the numerical optimizer it should be continuous and smooth, and comply with (2). The latter requirement forces us to use a different function than the exponential choice in [5]. We use the function

$$P(T_{req}, AT) = \left(\frac{AT}{T_{req}} \right)^q. \quad (7)$$

q determines how fast the penalty decays with separation, and hence how much separation is "enough", and k controls the importance of slack as compared to area. Note that (if feasible) our formulation still guarantees timing closure.

6. Results

We used the well-known MCNC benchmarks that come with SIS[6]. These circuits are mapped, and have a higher average fanout than may be expected at the macro level, making the problem harder. We used the tradeoff model of section 3.2.1, and formulated the programs with handcrafted timing constraints, and solved it with the commercial solver `MOSEK`[7]. This is a solver that exploits the convexness of our tradeoff modeling, and guarantees to find the optimum.

The results can be found in table 1. The first column contains the benchmark, the second and third contain information about the size of the problem, while the last three columns contain the runtimes for different formulations: forward, forward and pruned, and finally, backward and pruned. The best results are in bold. The experiments were conducted on a 1GHz, 512MB RAM PC running Linux. As expected, the results for the different formulations were identical, and runtimes were low considering the size of the problem.

Table 1: Results for different formulations.

Chip	#nodes	#paths	runtime		
			fw	fw pruned	bw pruned
C432	147	291e3	14s	17s	**5.3s**
C499	287	100e3	28s	**26s**	30s
C880	225	8442	13s	19s	**5.2s**
C1355	510	417e4	90s	**44s**	74s
C1908	349	196e3	47s	45s	**36s**
C3540	740	225e5	126s	114s	**76s**
C5315	1081	395e3	69s	**66s**	76s
C7552	1682	428e3	188s	130s	**69s**
C6288	2371	538e17	1493s	**528s**	1414s

In order to show how pruning influences the formulation of

[4] *Negative slack* means constraints are violated, *(positive) slack* means there is room to tighten a constraint, and *slack separation* is the difference between the most critical and other paths.

the program, we give average reductions in number of: variables: 47.8% (forward formulation), 35% (bw), constraints: 35.1% (fw), 20.2% (bw), and terms: 0.9% (fw) 10.2% (bw).

We also implemented the formulation for "robustness." In Fig. 6 we see an example slack histogram: at the price of only three percent area increase, the number of critical paths[5] reduces from 521 to 89.

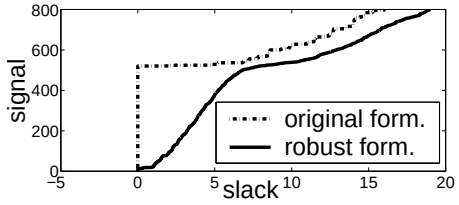

Figure 6: Three percent area increase reduces the number of critical paths dramatically.

7. Discussion

As can be seen from table 1, we can solve designs with up to 2371 modules within 528 seconds. As expected, there is always gain possible as compared to the unpruned forward formulation. Depending on the circuit structure either the forward pruned formulation or the backward pruned formulation is best.

Within a wireplanning flow, with its emphasis on stepwise refinement, the time budgets are golden. Through the tradeoff curves, they may be translated into areas, that have the status of *estimates*. These estimates may be used to guide top level floorplanning, or to evaluate the choice of footprint.

Since we do not give area budgets, and fixed delay synthesis is used, the calculated time budgets can always be met (at possibly high area price). With our robust formulation, slack may be used to alleviate the task of the synthesizer.

Contrary to classical static timing formulations, we maintain more than one or two variables at each node in the network. This is to account for all wire delay, and satisfy all timing constraints. As a side-effect, it is possible to extract sensitivities (Lagrange multipliers) from the solver for *all* critical pin-to-pin delays, instead of for the worst only, showing where timing is most likely hard to close, hence where most design effort should be focused.

As has been shown, depending on the network, there is a difference in efficiency for forward and backward formulations. For some parts of the network, forward formulation may be best, and for other parts the backward formulation. The parts can be "glued together" by timing constraints. For practical cases, the number of glue constraints outweighs the benefits from a "mixed formulation." Graphs with small cutsets in the middle, however, may be candidates where a mixed formulation is best.

[5]Here, a path is called *critical* if it has less than 2 time units positive slack.

8. Conclusions

For the first time, we have shown how to carry out time budgeting very early in a wireplanning flow. Early and accurate time budgeting has advantages such as the removal of iterations between time budgeting and tweaking, and the enabling of fully independent design of modules. We use a new static timing-like formulation that implicitly takes all (possibly billions of) pin-to-pin paths into account, and tradeoffs that can be obtained through a curve fitting technique. Total module area is minimized, wire delay is taken into account for all paths, and timing constraints are met. We also showed how make time budgets that are more "robust" to uncertainties downstream in the flow. As illustrated by our results, possibly billions of paths may exist in a given design. Our efficient formulation guarantees that design size is not a problem.

9. REFERENCES

[1] R.H.J.M. Otten and R.K. Brayton, *Planning for performance*, Proc. Design Automation Conference, p.122, June 1998

[2] R.H.J.M. Otten and R.K. Brayton, *Performance planning*, Integration the VLSI journal, vol 29, p.1

[3] A.R. Conn *et al.*, *Gradient-Based Optimization of Custom Circuits Using a Static-Timing Formulation*, Proc. Design Automation Conference, p.452, June 1999

[4] C. Visweswariah and A.R. Conn, *Formulation of Static Circuit Optimization with Reduced Size, Degeneracy and Redundancy by Timing Graph Manipulation*, Proc. International Conference on Computer-Aided Design, p.244, 1999

[5] X. Bai *et al.*, *Uncertainty-Aware Circuit Optimization*, Proc. Design Automation Conference, p.58, June 2002

[6] E. Sentovich *et al.*, *SIS: A System for Sequential Circuit Synthesis*, Technical Report UCB/ERL M92/41, Univ. of CA, Berkeley, May 1992

[7] `http://www.mosek.com`

[8] J. Grodstein *et al.*, *A delay model for logic synthesis of continuously-sized networks*, Proc. International Conference on Computer-Aided Design, p.458, 1995

[9] I. Sutherland *et al.*, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann publishers, November 1996

[10] R.B. Hitchcock *et al.*, *Timing analysis of computer hardware*, IBM Journal of Research and Development, p.100, 1982

[11] S.G. Abraham *et al.*, *Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs*, obtained through `http://www.hpl.hp.com/techreports/2000/HPL-2000-98.html`

[12] W. Gosti, *et al.*, *Wireplanning in Logic Synthesis*, Proc. International Conference on Computer-Aided Design, p.26, 1998